Introduction to Kernel Threads

This document is part of the HP-UX 11.x Software Transition Kit

Notices

The information contained in the HP-UX Software Transition Kit is subject to change without notice.

The name of Hewlett-Packard Company or the Hewlett-Packard logo may not be used in advertising or publicity pertaining to distribution of this information without specific, written prior permission. Hewlett-Packard Company makes no representations about the suitability of this information for any purpose. It is provided "as is" without express or implied warranty.

Hewlett-Packard disclaims all warranties with regard to this product, including all implied warranties of merchantability and fitness, in no event shall Hewlett-Packard Company be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this information.

HP has made every effort to ensure the accuracy of our product testing. However, because each customer's environment is different from HP's laboratory test environment, it is the customer's responsibility to validate the Year 2000 readiness of these products in their own environment. Therefore, information about the Year 2000 status of HP products is provided "as is" without warranties of any kind and is subject to change without notice. HP makes no representation or warranty respecting the accuracy or reliability of information about non-HP products. Such information, if any, was provided by the manufacturers of those products and customers are urged to contact the manufacturer directly to verify Year 2000 readiness. The information provided here constitutes a Year 2000 Readiness Disclosure for purposes of the Year 2000 Information and Readiness Disclosure Act.

HP encourages customers to be fully aware of the potential impact that the Year 2000 could have on their business environment and their ability to compete in the 21st century. HP also encourages its customers to take responsibility for addressing needed changes as quickly as possible.

Restricted Rights Legend.

Use, duplication, or disclosure by the U.S. Government Department of Defense is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause in DFARS 252.227-7013. Rights for non-DOD Government Departments and Agencies are as set forth in the Commercial Computer Software Restricted Rights clause, FAR 52.227-19 (c) (1,2).

Trademarks and Other Acknowledgments.

UNIX® is a registered trademark of the Open Group.

Adobe®, Acrobat®, and PostScript® are registered trademarks of Adobe Systems Incorporated.

Introduction to Kernel Threads

HP-UX 11.x includes "kernel threads" (the POSIX pthread library). The following discussion provides an introduction to kernel threads and their use. For more information, see the book on which this discussion is based: "Thread Time - The Multithreaded Programming Guide", by HP's Scott Norton and Mark DiPasquale (HP Professional Book, published by Prentice Hall, ISBN 0-13-190067-6).

This document contains the following sections:

What is a Thread?

A thread (as defined by Maurice Bach's "The Design of the UNIX Operating System") is an independent flow of control within a process, composed of a context (including register set and a program counter) and a sequence of instructions to execute. The traditional flow of control within a program has been a process with a single context of registers, following a single path through the code. This is referred to as the "process-based" or "single-threaded model". For an application in a single-threaded model to handle multiple tasks, it would have to break those tasks up into multiple processes, coordinate with signal handlers to provide some concurrency, or simply deal with those tasks serially (one after the other).

A multi-threaded model for a process allows multiple concurrent paths of execution, or "threads", through the same process, allowing a single process to handle multiple tasks without having to program to serially move from one to the other. Those familiar with HP-UX internals will additionally note that a multi-threaded process shares most of the process-related data, with only an additional kthread structure allocated for each of the multiple threads to give its register context, along with a private stack area and related resources. Access to multi-threading functionality is a tool a programmer can choose to use to improve the coding style or throughput of an application.

Impact to the Non-Programmer

Although most of the changes in the multi-threaded programming environment affect only the programs written in this way, some of this will be visible to the system administrator as well. In their first release, these changes will mean seeing single processes in applications which used multiple processes in earlier releases. As tools evolve to be more "thread-aware", one can expect

performance and system-monitoring tools to give more thread-specific information in the place of process-specific information in current versions, where appropriate.

Why Program with Threads?

A multi-threaded application can have several advantages over a single-threaded application. Among these are:

- more natural programming style: imagine an application which has several disjoint tasks such as doing computations, reading input from a socket, refreshing the GUI, etc. In a single-threaded model, the application must take on the responsibility of switching between these various tasks, but in a multi-threaded application, they can be set up to operate mostly independent of one another.

- the ability to gain concurrency: a single-threaded application must do one task, wait for completion, then do the next one. For example, in an application which must read data, modify it, and write it back out, a single-threaded application may spend much of its time blocked on I/O. A multi-threaded version of the same application can gain concurrency by scheduling the next read while the previous write is completing.

- the ability to parallelize an application: in a computationally-intensive application, it would be advantageous to divide the task among the various processors on a multiple-processor system. The single-threaded model cannot do this within the process, as the process is scheduled as a single unit, and therefore on only one processor. A multi-threaded application could run different threads of the same process in parallel across the multiple processors.

Note that it is possible to gain the benefits of concurrency on either a uniprocessor or a multiprocessor system, but parallelism can only be achieved on multi-processor system. In both concurrent and parallel applications, however, performance or throughput improvements can often be achieved by multi-threading.

User, Kernel, Bound, and Unbound Threads

Because of the evolution of the various threads models, talking about threads can be confusing just because of the variations in terminology. It is important to clarify the difference between user and kernel threads. User threads are those which the application creates, and kernel threads are those which the kernel can "see" and schedule. A user application can implement a multi-threaded application without kernel threads by implementing a user-space scheduler to switch between the various threads for the process. Since this exists only in user-space, the kernel still sees only a single thread -- these threads are therefore referred to as unbound, since they do not correspond to a thread the kernel can see and schedule. If each of these threads are bound to a single kernel thread, there is no need for the user-space scheduler, and these threads are referred-to as bound. Notice that only *bound* threads (those which the kernel can see) can give the benefits of parallelism on a multi-processor machine. It is the implementation of kernel threads which is new on 11.00.

How is This Different from Dividing My Application into Multiple Processes?

A process contains much more than just a register context, and therefore is fairly expensive to create and destroy in comparison to a thread. Also, when sharing information, the programmer must take into account the management of special shared memory areas and inter-process communication facilities, whereas a multi-threaded process is running several threads amongst the same shared resources, making it easier for data sharing between them, and for the relatively inexpensive creation and destruction of threads as needed.

What are the Specifics of HP's Threads Implementation?

The implementation on 11.00 brings HP-UX in line with the POSIX 1003.1c standard, which includes the standard POSIX interface (1003.1), the POSIX real-time scheduling standard (1003.1b) and the implementation of kernel threads (1003.1c). In addition, HP-UX 11.00 implements several thread extensions, including those which are defined by other standards bodies, such as X/Open, and those which are unique to HP at this time. Included in HP's extensions is our first officially-supported multi-processor "processor affinity" functions, which allow a thread to bind itself to a processor and query information about the number of processors on the system and which processors are running which threads. The number of threads per process is bounded by the kernel tunable parameters max_thread_proc (maximum threads per process, default 64 / max nkthread) and nkthread (number of threads total on the system, default roughly twice nproc / max of 30000).

How is this related to CMA or DCE threads that were available prior to HP-UX 11.00?

The CMA library (distributed as a part of DCE) is a user-space scheduler (therefore uses only unbound threads), and is based on an earlier draft of the pthreads library standard. This sort of implementation can be referred-to as an "M x 1" model, since multiple user threads share one kernel thread. The POSIX pthread library available in 11.x is a "1 x 1" model, since each user-space thread is bound to a single kernel thread. Since the CMA implementation uses only unbound threads, it is not possible for it to gain the benefits of running the different threads of a process on different processors in a multi-processor system. In its favor, the M x 1 implementation has some advantages due to the fact that it resides entirely in user space: for example, the overhead of creating and destroying threads is much lighter than the 1 x 1 implementation. Also, its administration is a little simpler: for example, a CMA-based multi-threaded application can create as many threads as memory will allow, whereas the maximum number of threads on the system and per process are kernel tunable parameters which may need to be changed. Further, the CMA implementation was based on an earlier draft of the standard, so applications written for DCE/CMA threads will not be able to utilize the new kernel threads functionality in 11.x without some modification to the application, and recompilation with the new pthread library.

Programming a Multi-Threaded Application

Every process begins as a single-threaded entity, and can create additional threads using the pthread_create(3T) library call, providing it the function which that thread should execute. An individual thread later exits with a call to pthread_exit(3T), or by returning from its initial function. Note that a call to exit(2) by any thread will immediately cause the entire process (all threads) to exit. The return codes from exited threads can be collected by calling pthread_join(3T), which can therefore also be used to detect when a thread has exited.

These functions, as well as other pthread-related functions are found in the pthread library, which is provided at 11.00 (compile with the -lpthread flag). Other facilities which are provided by this library include functions to manage and use threads themselves, facilities for synchronizing access to shared data (such as mutual-exclusion "mutex" locks, read-write locking and POSIX-standard semaphores), for synchronizing the work of the various threads (such as with condition variables and inter-thread wakeup-facilities), and for modifying the system's scheduling policies to determine the relative priorities of the various threads within the process and other processes on the system. As you might imagine, the bulk of the work when designing a multi-threaded application is in ensuring that the multiple concurrent paths through the code do not collide with each other and cause confusion or data corruption. Similarly, the programmer takes on the additional responsibility of ensuring that the synchronization and protection mechanisms are used properly to avoid synchronization problems which could lead to deadlocks or data corruption.

Many different programming and synchronization models are discussed in "Thread Time" and other programming books. Please refer to these for ideas on how to design a multi-threaded application, and to the online man pages of the various pthread_mutex*, pthread_rwlock*, pthread_cond*, sem_* and sched_setscheduler functions for their usage.

Handling Signals in a Multi-Threaded Environment

Signals handling can get quite complex within a multi-threaded application. Each thread has its own signal mask, but the signal handlers are installed on a per-process basis (i.e. all threads have the same handlers). Additionally, depending on the type of signal, and how it gets delivered, a signal can be delivered to either a thread or the process as a whole. This can be a problem when multiple threads are blocked on the same signal, especially if one thread starts an event which triggers a signal which is received by a different thread. In order to reduce the complexity involved, the pthreads library provides a method for one thread to signal a specific thread (pthread_kill), and to have a particular thread handle the receipt of a signal (sigwait). In this model, a thread can be dedicated to handling a particular set of signals, freeing the other threads from having to handle the interruptions. By dedicating a thread to the handling the signals, and having the other threads block these signals, this problem can be greatly simplified.

What Is A "Thread-Safe" Library?

A completely thread-safe library is one which can be called by multiple threads of a process without any coordination between them. There are also partially-thread-safe library routines

which can be called by multiple threads, but require the application to coordinate the usage of the library between the different threads. Completely non-thread-safe library routines cannot be used in multi-threaded application. Programmers should always consult library function documentation to see the level of thread safety provided when using multiple threads in their applications.

Do I Have to Modify My Application to Run on the Multi-Threaded Platform?

The short answer is no: a single-threaded application will continue to run on HP-UX 11.x as it had in the past, using only one kernel thread per process.

If your application was written using CMA/DCE threads, you will need to evaluate whether you want to continue using that programming interface, with a user-space scheduler and single kernel thread, or to convert to the newer pthread interface to take advantage of the kernel-based thread functionality now available. Note that that both user-space and kernel-space schedulers have advantages and disadvantages, so the relative merits of both both should be weighed carefully.

Those considering multi-threading their single-threaded applications should carefully consider the benefits of multi-threading to their application design, and weigh this against the added complexity of managing shared data between multiple threads.

How Do I Debug a Multi-Threaded Application?

With multiple concurrent paths of execution within the same program, multi-threading adds a great deal of complexity to the debugging. To aid in this, the dde debugger has included thread-specific functionality to trace the multiple threads within an application. Note that this is different from the previously-available CMA threads functionality, which is still available in dde through a command-line option. The now obsolescent xdb debugger does not have any thread-related functionality, and therefore will show only one thread in a process -- whichever hit the breakpoint or triggered the core being examined.